EE 341 – Lab 1

Elementary Music Synthesis

Lab Objectives

The purpose of this lab is to get yourself familiarized with constructing and operating on discrete-time signals under the Python + SciPy environment. You will also gain some understanding of the physical meaning of the signals by using audio playback. Specifically, we will first help you setup a Python development environment on your computer which you will be using throughout the quarter. Then, we will guide you through the basics of Python and practice how to use it to process signals. Finally, to demonstrate your understanding, you will need to finish 3 tasks that generate audio signals of increasing complexity.

Python Introduction

Setup

If you haven't had a Python environment ready on your computer yet, we recommend you install Anaconda, a Python/R distribution bundled with many scientific computation packages. You can download it from the official website: https://www.anaconda.com/download/. Make sure to download the **3.7** version that matches your OS.

After the installation is finished, launch the **Anaconda Prompt**. If you are using Windows, you can find it by searching "Anaconda" in the Start Menu. If you are using MacOS, open Launchpad and click the Terminal icon. Now enter "pip install simpleaudio" (without the quotation marks) in the prompt. This will install the audio playback package that we will use later. If you are using some other Python distribution that is not Anaconda, make sure to also install "scipy", "matplotlib" and "jupyter" using the "pip install" command. These packages are provided along with Anaconda so you don't need to manually install them if you are using Anaconda.

Working with Jupyter Notebook

Jupyter Notebook is a web-based application that allows you to write formatted text and executable Python program in the same document. In this way, your code and your lab report can be combined together and submitted as a single file. To launch Jupyter Notebook, simply type "jupyter notebook" (without the quotation marks) in the command prompt. After hitting enter, a web-page that shows your home directory should appear on your browser. Create a new folder using the "New" button on the top-right corner. An "Untitled Folder" should appear on the list. Rename it to "ee341lab" by clicking on the checkbox before it and then on the "Rename" button near the top of the page. This folder will be used to store all your works on the labs. Enter the folder and create a new Python 3 notebook using the "New" button. Rename the notebook you just created to "lab1" by clicking on the current title "Untitled" on the top of the screen.

Each Jupyter Notebook document is comprised of several cells. A cell can either be code (Python code to be executed) or text (written in Markdown format). By default, new cells are code cells. The box with "`In [ ]`" on your newly created document is a code cell. The type of a cell can be changed using the drop-down menu on the toolbar. New cell can be created by clicking the plus icon on the toolbar. Running a code cell will execute the Python code in it and show the output. Running a text cell will render the formatted text.

Developed/revised by Yibo Cao, Kamil Jiwa, and Tai Chen 2019

Python Quickstart

Just like the Java programming language you have learned in CSE 142/143, Python is a general-purpose programming language that shares many of the same basic concepts. This section will quickly walk you through how you can express those concepts in Python. Let's start with an example program:

```
# This is a comment.

my_integer = 5
my_float = 3.3
my_string = "test"

print("my_integer is " + str(my_integer))
```

To run the program, copy it to a code cell and click "Run" or hit Ctrl + Enter. As can be seen from the program, variables don't need to be declared before use and thus they don't have type specifications. However, values do have types. That's why `my_integer` in the print statement has to be casted to string using the `str` function before concatenating with another string, as concatenating string directly with integers causes error.

Here is another example program that shows the control structures in Python:

```
for i in range(10):
  print(i)
  if (i % 2 == 0):
    print("even")
  else:
    print("odd")

print("after loop")
print(i)
```

As shown by the example, control structures in Python use indentations instead of curly braces to specify the start and the end of the body. Also notice that control structures in Python doesn't have scopes: the loop variable `i` is still visible after the end of the for-loop.

In Python, functions can be defined using the `def` keyword as shown below. Similar to control structures, the start and the end of the function body is specified by the level of indentation. When calling a function, the arguments can be explicitly specified by their names.

```
def square(x):
  return x * x

print(square(2))
print(square(x = 4))
```

Using NumPy

NumPy, a part of the SciPy project, is a Python package that provides a powerful n-dimensional array (ndarray) datatype and many useful functions that works on ndarrays. Since discrete-time signals are implemented in computers as arrays, a well understanding the NumPy package is essential for fluently manipulating signals in Python. To use NumPy, add the following statement to the beginning of a program:

```
import numpy as np
```

This imports the `numpy` package and alias it as the shorthand `np`. To start with NumPy you need to create some ndarrays. There are several ways to do so: some commonly used ones are listed below. Try each of them on your Jupyter Notebook and examine the outputs.

- Array can be created by directly specifying the elements. For example:
  `np.array([1, 3, 5, 7])`
  This creates an array of 4 elements: 1, 3, 5 and 7.
- It is common to create an array of a fixed number of zeros / ones. For example:
  `np.zeros(10)`
  This creates an array of 10 zeros.
  `np.ones(10)`
  This creates an array of 10 ones.
- Array can also be created by generating evenly spaced values in a range. For example:
  `np.arange(5, 25, 2)`
  This creates the array [5, 7, 9, 11, 13, 15, 17, 19, 21, 23]. The first argument of `arange` is the start of the range (inclusive), the second argument is the end of the range (exclusive) and the last argument is the step size.
- Another way of generating evenly spaced values is to specify the total number of numbers to generate instead of the step size. For example:
  `np.linspace(0, 100, 11)`
  This creates the 11 numbers from 0 to 100: [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]. Note that the end of the range is inclusive. To make it exclusive, add the argument `endpoint = False`.

Now you have created some ndarrays, but what can you do with them? Certainly, you can access the individual elements in the style "`array[index]`" just like in Java. However, NumPy provides many far more efficient ways of doing computations with ndarrays. Some of the commonly used ones are shown in the example program below. Try them out on your Jupyter Notebook and examine the outputs.

```
# Create some arrays.
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Arithmetic operators can be applied to arrays of the same size.
# The operation is performed element-wise.
# For example, the expression below computes [1+4, 2+5, 3+6].
print(a + b)

# When an array is operated with a number, the same operation with the number
# is performed on all of the array's elements. This is known as "broadcasting".
# For example, the expression below computes [1*2, 2*2, 3*2].
print(a * 2)

# Commonly used mathematical functions (abs, sqrt, ...) are provided by NumPy.
# When they're applied to an array, the operation is performed on each element.
# For example, the expression below computes [sin(1), sin(2), sin(3)].
print(np.sin(a))

# Concatenate a and b. Note that the argument itself is an array.
print(np.concatenate([a, b]))

print(np.sum(a))  # Computes the sum of all elements in a.
```

```
print(np.max(a))  # Computes the maximum of all elements in a.
print(np.mean(a)) # Computes the mean of all element in a.
print(len(a))     # Gets the total number of elements of a.

# To access a sub-range of an array, the slicing operator "array[start:end]"
# can be used. For example:
c = np.array([1, 3, 5, 7, 9])
print(c[0:3]) # Get the first 3 elements of c.
print(c[2:])  # Get all elements of c except the first two.

# A slice of an array can just be used as a regular array.
# For example, we can't add b and c because they have different sizes.
# However, we can add b and a slice of c:
print(b + c[0:3]) # Computes [4+1, 5+3, 6+5]
```

The examples above cover most of your needs. However, if you ever need to perform some operations on ndarrays that is not covered in the examples, you can always refer to the documentations in NumPy's official website https://docs.scipy.org/doc/numpy/user/index.html for a comprehensive tutorial.

Plotting Data

Plotting data in Python is usually done using the matplotlib package. To import the package, put the following two lines after your NumPy import statement:

```
%matplotlib notebook
import matplotlib.pyplot as plt
```

To plot a single curve on a single plot, use the following example:

```
x = np.arange(0, 10, 1/32) # x axis data
y = np.sin(x)              # y axis data
plt.plot(x, y)          # plot the data
plt.title('y=sin(x)')   # set the title
plt.xlabel('x')         # set the x axis label
plt.ylabel('y')         # set the y axis label
plt.xlim(-1, 11)        # set the x axis range
plt.ylim(-2, 2)         # set the y axis range
plt.grid()              # enable the grid
```

To plot multiple curves on a single plot with a legend, use the following example:

```
x = np.arange(0, 10, 1/32) # x axis data
y1 = np.sin(x)             # y axis data 1
y2 = np.cos(x)             # y axis data 2
y3 = np.tan(x)             # y axis data 3
plt.figure(1)              # create figure 1
plt.plot(x, y1, label='sin(x)')
plt.plot(x, y2, label='cos(x)')
plt.plot(x, y3, label='tan(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.xlim(-1, 11)
plt.ylim(-2, 2)
plt.suptitle('Trigonometric Functions')
plt.legend()
plt.show()
```

To create multiple subplots in the same figure, use the following example:

```
x = np.arange(0, 10, 1/32) # x axis data
y1 = np.sin(x)             # y axis data for subplot 1
y2 = np.cos(x)             # y axis data for subplot 2
y3 = np.tan(x)             # y axis data for subplot 3

plt.figure(2)              # create figure 2

plt.subplot(3, 1, 1)       # (number of rows, number of columns, current plot)
plt.plot(x, y1)
plt.title('sin(x)')
plt.xlabel('x')
plt.ylabel('y')

plt.subplot(3, 1, 2)
plt.plot(x, y2)
plt.title('cos(x)')
plt.xlabel('x')
plt.ylabel('y')

plt.subplot(3, 1, 3)
plt.plot(x, y3)
plt.title('tan(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.ylim(-1, 1)
```

Now you have learned the fundamentals of dealing with array representations of discrete time signals in Python. To demonstrate your knowledge, you need to independently complete and turn-in the following 3 tasks. While you are writing your Python implementations for the tasks, make sure to also document your design in text cells surrounding your code segments. Remember that the Jupyter Notebook you're working on will eventually develop into a lab report. Don't forget to frequently save your progress.

Note that the text cells in Jupyter Notebook follows the Markdown syntax, which uses symbols to control the formatting. For example, surrounding texts with a pair of ** will make the text bold. If you are not familiar with Markdown, follow this link for a tutorial.

Task 1: Generating Musical Notes

Background

For this task, we explore how to use simple sinusoidal waves to realize musical notes and how to bring them together to compose a segment of music. At the end of this task, you will construct the first few measures of the song Scarborough Fair.

There are seven natural notes: A, B, C, D, E, F and G. After G, we begin again with A. Music is written on a "staff" consisting of five lines with four spaces between the lines. The notes on the staff are written in alphabetical order. The first line is E as shown in Figure 1. Notes can extend above and below the staff. When they do, ledger lines are added.

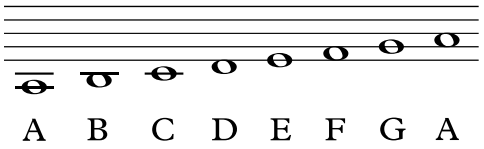

Figure 1 Natural Notes

Musical notes are arranged in groups of twelve notes called octaves. The notes that we'll be using for Scarborough Fair are in the octave containing frequencies from 220 Hz to 440 Hz. The twelve notes in each octave are logarithmically spaced in frequency, with each note being of a frequency $2^{1/12}$ times the frequency of the note of lower frequency. Thus, a 1-octave pitch shift corresponds to a doubling of the frequencies of the notes in the original octave.

Table 1 shows the ordering of notes in the octave to be used to synthesize the opening of Scarborough Fair, as well as the fundamental frequencies for these notes.

Table 1: Notes in the 220 – 440 Hz Octave

Note	Frequency
A	220
A♯, B♭	$220 \times 2^{1/12}$
B	$220 \times 2^{2/12}$
C	$220 \times 2^{3/12}$
C♯, D♭	$220 \times 2^{4/12}$
D	$220 \times 2^{5/12}$
D♯, E♭	$220 \times 2^{6/12}$
E	$220 \times 2^{7/12}$
F	$220 \times 2^{8/12}$
F♯, G♭	$220 \times 2^{9/12}$
G	$220 \times 2^{10/12}$
G♯, A♭	$220 \times 2^{11/12}$
A	440

A musical score is essentially a plot of frequencies (notes) on the vertical scale versus time (measures) on the horizontal scale. **The musical sequence of notes for the piece you will synthesize is given in Figure 2.** The following discussion identifies how musical scores can be mapped to tones of specific pitch and duration.

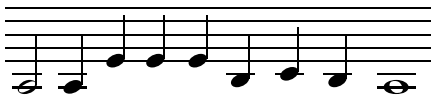

Figure 2: Musical Score for Scarborough Fair

In the simplest case, each note may be represented by a burst of samples of a sinusoid followed by a shorter period of silence (samples of zeros, which are a pause). The pauses allow us to distinguish between separate notes of the same pitch. The duration of each note burst is determined by whether the note is a whole note, a half note, or a quarter note (see In this Lab, use a duration of **4,000** samples for 1 count.

Figure 3).

Whole note: it lasts 4 counts.

Half note: it lasts 2 counts.

Quarter note: it lasts 1 count.

In this Lab, use a duration of **4,000** samples for 1 count.

Figure 3: Types of Notes

Therefore, your whole notes should be four times the duration of your quarter notes. The short pause you use to follow each note should be of the same duration regardless of the length of the note. Longer periods of silence that are part of the musical score are indicated by one or more rest symbols. There are no rest symbols in the score you are given.

Note that A – G only yields seven notes; the additional notes are due to changes in pitch called sharps (denoted by the symbol ♯) or flats (denoted by the symbol ♭) that follows a given note. A sharp increases the pitch by $2^{1/12}$ and a flat decreases it by $2^{1/12}$. There are no sharp or flat symbols in the music you are going to create.

In the musical score in Figure 2, the first half note and quarter note are both A. The next three quarter notes are all E and so on. You can get the fundamental frequencies for these notes from Table 1.

Assignment 1

Write a Python program that generate the audio samples of the song using a sampling rate of **8 kHz**. To play the samples, firstly add `import simpleaudio as sa` after all your existing imports. Then add the following statement at the end of your program. Replace "`samples`" with the name of the variable that stores the generated audio samples.

```
sa.play_buffer((samples * 32767).astype('int16'), 1, 2, 8000).wait_done()
```

This statement assumes the amplitude of each sample is in the range [-1, 1], otherwise the audio will be distorted. Before playing the sample, the amplitude is magnified by 32767 ($2^{15} - 1$) so that it is audible through your speakers. The samples must also be converted from a NumPy array into a list of 16-bit integers by using the `astype` method. Lastly, the final three arguments to `play_buffer` are the number of channels (one in this lab), the number of bytes per channel (two, since each byte is 8 bits), and the sample rate.

Guide

- **How to generate a note**
  It may seem trivial, but probably the best point to start this Lab is by writing some Python code that will generate and play a note of music. In this Lab, a musical note is a sinusoid of a certain frequency (which?), a certain duration (how many seconds, or how many samples?), and sampled at a certain sampling rate (which one?). To create the sinusoid in Python, you need to choose values for the three parameters. You can then use those parameters to generate an ndarray that you can use as input to the `np.sin` function (recall the examples provided in the Using NumPy section). Once you've created the sinusoid, you can play it using the `sa.play_buffer` function shown above in the Assignment section.
- **How to concatenate notes and rests**
  Now that we know how to create a note, the next step is to concatenate a series of notes, separated by short rests. Recall that `np.concatenate([a, b, c])` joins the arrays a, b and c together. The rest

between notes is just some silence, that you can generate with the `np.zeros` function. The amount of rest doesn't matter much, experiment a little bit to see what sounds ok (hint, it doesn't have to last longer than a quarter of a second, and probably a lot less).

- **Using a for-loop may save you time**
  When creating the whole song, you can of course write the entire concatenation of all the notes and rests as one long Python statement. But it may be easier and save you time if you used a for-loop for it. In that case, you would need to store the frequency and duration of each note in one (or two) array(s), and then do something like this:

```
rest = ...            # define the rest between notes here
song = np.array([])   # start with an empty song
for i in range(N):
  thisNote = generateNote(frequency[i], duration[i], samplingRate)
  song = ...          # concatenate this note and a rest to the song
```

  Of course, you need to properly define all the variables and arrays, write the `generateNote` function and fill in some blanks here.

Task 2: Volume Variations

There are many ways of improving the perceived quality of a synthesized sound. In this task you will learn about one such method: varying the note volume over time.

Typically, when a note is played, the volume rises quickly from zero and then decays over time, depending on how hard the key is struck and how long it is depressed. The variation of the volume over time is divided into four segments: **Attack, Decay, Sustain, and Release (ADSR).** For a given note, volume changes can be achieved by multiplying the sinusoid by another function called an envelope function. An example of such function is shown in Figure 4.

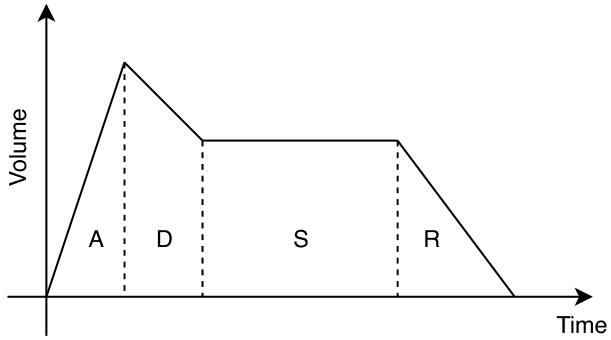

Figure 4: An ADSR Envelope

Assignment 2

Improve the quality of the sound you produced in Task 1 by applying an ADSR envelope to each note. Do not directly modify your code for Task 1. Instead, copy it over and modify the copy so that you have both your original code and the improved code in your notebook. You don't need to copy the functions or constants (e.g. the frequency / duration array) you defined in Task 1 if you don't plan to modify them.

Guide

- **How to apply an envelope to a note**
  To improve the quality of the note, we apply an ADSR envelope to it. This consists of 2 steps: first we must construct the envelope, and then we must apply it to the note. The envelope is an ndarray that has the same length as a note, and that has values between 0.0 and 1.0, indicating the volume of each sample of the note. To generate the linear parts of the envelope, you may find the `arange` function or the `linspace` function helpful (refer back to the Using NumPy section for example usages). It's probably easiest to generate each linear piece first, and then concatenate them to form the whole envelope. To apply the envelope to the note, simply element-wise multiply the note ndarray and the envelope ndarray. Remember that for element-wise operation to work, the two ndarrays must have the same size. **Question:** instead of enveloping with the ADSR function, how would you envelope with a decaying exponential? Include your answer in the notebook.

Task 3: Overlapping Notes

Another improvement in perceived quality can be achieved by overlapping some notes as done by advanced piano players. As the volume of one note is decaying, another note is played. Mathematically, this can be accomplished by allowing the time regions occupied by subsequent sinusoids to overlap. This will yield a much smoother, less staccato-sounding piece.

Assignment 3

Improve the quality of the sound you produced in Task 2 by allowing the end of one note and the beginning of the next note to overlap slightly in time. Again, work on a copy instead of directly modifying your original Task 2 code. Keep in mind that the samples of your final song should not exceed the ± 1.0 range. If you are not sure, you can double check your waveform by plotting it.

Guide

- **How to create overlapping notes**
  The last improvement to the quality of the song is to use overlapping notes. **Question:** Would you still use concatenation of notes to accomplish this? If so, how? If not, why not? How would you do it instead? (Hint: You can mix two signals by adding them together. So, if you create one signal that is one note followed by a bunch of zeros, and another signal that is a bunch of zeros followed by another note, what would the result be? How would you use this technique to construct the entire song?).

Lab Submission Requirements

Submit your completed Jupyter Notebook in the `ipynb` format to Canvas. The notebook file can be obtained by clicking the menu: File – Download as – Notebook or directly accessed from the ee341lab folder in your home directory. Make sure to include the code of all the 3 tasks in code cells. In your notebook, briefly describe your results and discuss the problems you encountered, the solutions that you came up with and any choices you made regarding to the length of rests, details of the envelope, etc. Also, answer the questions asked in the Guide sections.